50272-101

REPORT DOCUMENTATION PAGE	1. REPORT NO. DCA/SW/MT-88/001m	2.	3. Recipient's Accession No.
4. Title and Subtitle: Defense Communications Agency Upper Level Protocol Test System Simple Mail Transfer Protocol Remote Driver Specification			5. Report Date: May 1988 6.
7. Author(s)			8. Performing Organization Rept. No.
9. Performing Organization Name and Address: Defense Communications Agency, Defense Communications Engineering Center, Code R640, 1860 Wiehle Ave., Reston, VA 22090-5500			10. Project/Task/Work Unit No. 11. Contract(C) or Grant(G) No. (C) (G)
12. Sponsoring Organization Name and Address			13. Type of Report & Period Covered: FINAL 14.

15. Supplementary Notes

For magnetic tape, see: ADA195128

16. Abstract (Limit: 200 words)

This document is part of a software package that provides the capability to conformance test the Department of Defense suite of upper level protocols including: Internet Protocol (IP) Mil-Std 1777, Transmission Control Protocol (TCP) Mil-Std 1778, File Transfer Protocol (FTP) Mil-Std 1780, Simple Mail Transfer Protocol (SMTP) Mil-Std 1781 and TELNET Protocol Mil-Std 1782.

17. Document Analysis a. Descriptors

Protocol Test Systems
Conformance Testing
Department of Defense Protocol Suite

b. Identifiers/Open-Ended Terms

Internet Protocol (IP)
Transmission Control Protocol (TCP)
File Transfer Protocol (FTP)
Simple Mail Transfer Protocol (SMTP)
TELNET Protocol

c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Unlimited Release	UNCLASSIFIED	35
	20. Security Class (This Page) UNCLASSIFIED	22. Price

(See ANSI-Z39.18) See Instructions on Reverse OPTIONAL FORM 272 (4-77) (Formerly NTIS-35) Department of Commerce

DEFENSE COMMUNICATIONS AGENCY

UPPER LEVEL PROTOCOL TEST SYSTEM

SIMPLE MAIL TRANSFER PROTOCOL MIL-STD 1781
REMOTE DRIVER SPECIFICATION

MAY 1988

Disclaimer Concerning Warranty and Liability

This software product and documentation and all future updates to it are provided by the United States Government and the Defense Communications Agency (DCA) for the intended purpose of conducting conformance tests for the DoD suite of higher level protocols. DCA has performed a review and analysis of the product along with tests aimed at insuring the quality of the product, but does not warranty or make any claim as to the quality of this product. The product is provided "as is" without warranty of any kind, either expressed or implied. The user and any potential third parties accept the entire risk for the use, selection, quality, results, and performance of the product and updates. Should the product or updates prove to be defective, inadequate to perform the required tasks, or misrepresented, the resultant damage and any liability or expenses incurred as a result thereof must be borne by the user and/or any third parties involved, but not by the United States Government, including the Department of Commerce and/or The Defense Communications Agency and/or any of their employees or contractors.

Distribution and Copyright

Comments

Comments or questions about this software product and documentation can be addressed in writing to:

DCA Code R640
1860 Wiehle Ave
Reston, VA 22090-5500
ATTN: Protocol Test System Administrator

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

APPENDICES

SECTION 1 - SCOPE AND PURPOSE

This manual describes the Simple Mail Transfer Protocol (SMTP) Remote Driver (RD) for the protocol test system. Section 2, "The Protocol Test System," summarizes the testing procedures used by the protocol laboratory. Section 3, "Remote Driver Functions," contains guidelines for developing a RD for SMTP where an implementation under test (IUT) resides. In addition, Section 3 describes the commands this RD must interpret and execute to be capable of SMTP testing. The response packets the RD must send back to the Central Driver (CD) are also specified.

SECTION 2 - THE PROTOCOL TEST SYSTEM

The protocol test system (PTS) exercises an IUT performing peer protocol exchanges with a reference implementation. A driver controls each peer protocol through its upper level interface. To generate reproducible results, a script controls each driver.

The major components of the PTS are:

- Central Driver (CD) -- To coordinate and monitor protocol testing;
- Remote Driver (RD) -- To exercise the protocol IUT and provide communication links with the CD;
- Laboratory Slave Driver (LSD) -- To exercise the Protocol Reference Implementation, record protocol data units exchanged over the test connections, and provide links with the CD;
- Test Scenario Language (TSL) -- To enable a tester to specify standardized scripts;
- Scenario Language Compiler and Libraries -- To produce code for automated testing; and
- Protocol Reference Implementation -- To define standard functions for the level being tested.

The CD, which coordinates and monitors protocol testing, combines the input it receives from the LSD and the RD to determine the success or failure of each test.

The RD receives protocol command codes from the CD, translates and issues protocol commands to the IUT, and sends back protocol responses from the IUT to the CD. Driver commands control the RD process itself and do not affect the IUT's state. Protocol commands that the RD passes to the IUT as SMTP commands have specific parameters

and directly affect the IUT's state. Figure 3-2 diagrams the flow of commands and data between the drivers and their peer protocols. Section 3 explains in detail the command codes and the format of the data.

SECTION 3 - REMOTE DRIVER DESIGN AND PROTOCOL TESTING

The following subsections describe the Remote Driver (RD) design and explain how all drivers interact to communicate protocol commands or responses in lab testing.

3.1 THE COMMAND CHANNEL

All drivers except the RD reside at the protocol laboratory. The RD operates as a background process using a transport level connection, which links the RD at a remote site with the Central (CD), the laboratory drivers, and the reference and IUT protocols. The transport protocol in use is the Transport Control Protocol (TCP).

To set up the command channel, the RD issues a passive open on a well-known port and waits for the Central Driver (CD) to perform an active open to that port. The RD functions as server and listens for further connection requests from the CD on the well-known port after the command channel has been established. The command channel is ready when the transport connection to the RD and slave drivers has been established, and all drivers have initiated communications with their respective protocols.

Figure 3-1 is a diagram of connection establishment, and Table 3-1 (p. 3-3) gives destinations and port numbers. Appendix C includes an example of command channel establishment implemented in UNIX/C (Figure C-1).

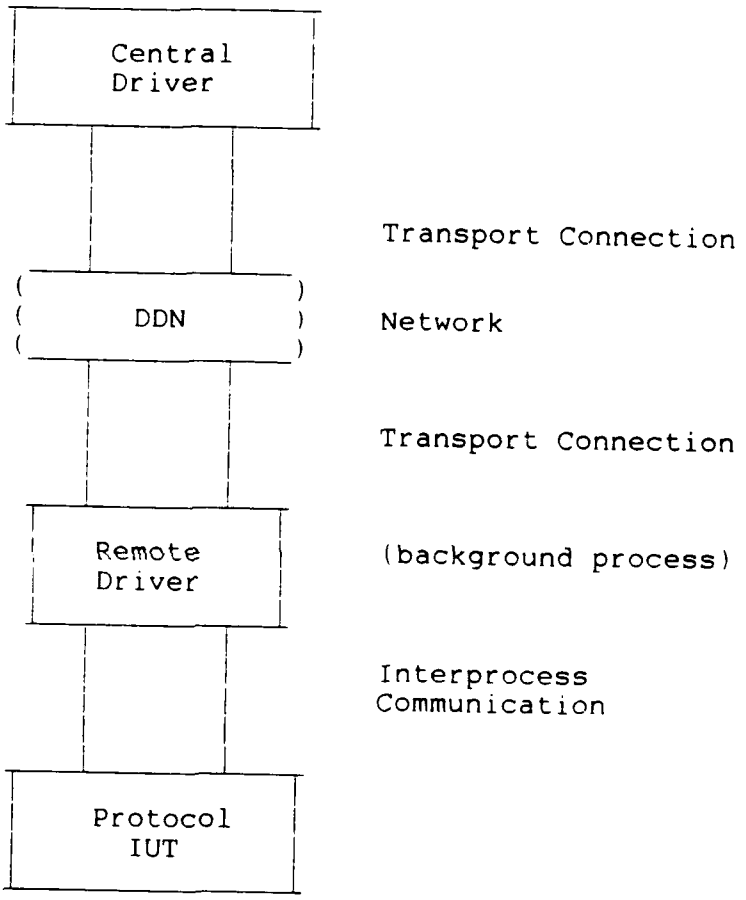

Figure 3-1. Connection Establishment

Table 3-1. Destinations and Port Numbers

Destination	Port Number
RD Passive Open Port	1040
SMTP IUT Receiver Passive Open Port	25

3.2 FLOW OF COMMANDS

Packets containing protocol or driver command codes are passed over the command channel from the CD to the RD (Figure 3-2, p. 3-4; Table 3-3, p. 3-11). The RD translates these codes into the appropriate commands, then issues the commands to the protocol IUT. Similarly, responses from the protocol IUT are received by the RD and forwarded over the command channel to the CD (Figure 3-3, p. 3-5). The CD determines from the combined test responses whether the IUT has functioned as expected and reacts accordingly, by taking the next appropriate action in the testing process. The flow of commands is then repeated.

3.3 INPUTS AND OUTPUTS

Data are transmitted from the testing facility to the protocol laboratory in packets. The laboratory implementation of the RD uses a Packet Assembler/Disassembler (PAD) function to control the input and output of these packets.

The RD must be able to receive packets from the CD, translate number codes into commands, and interpret ASCII strings of

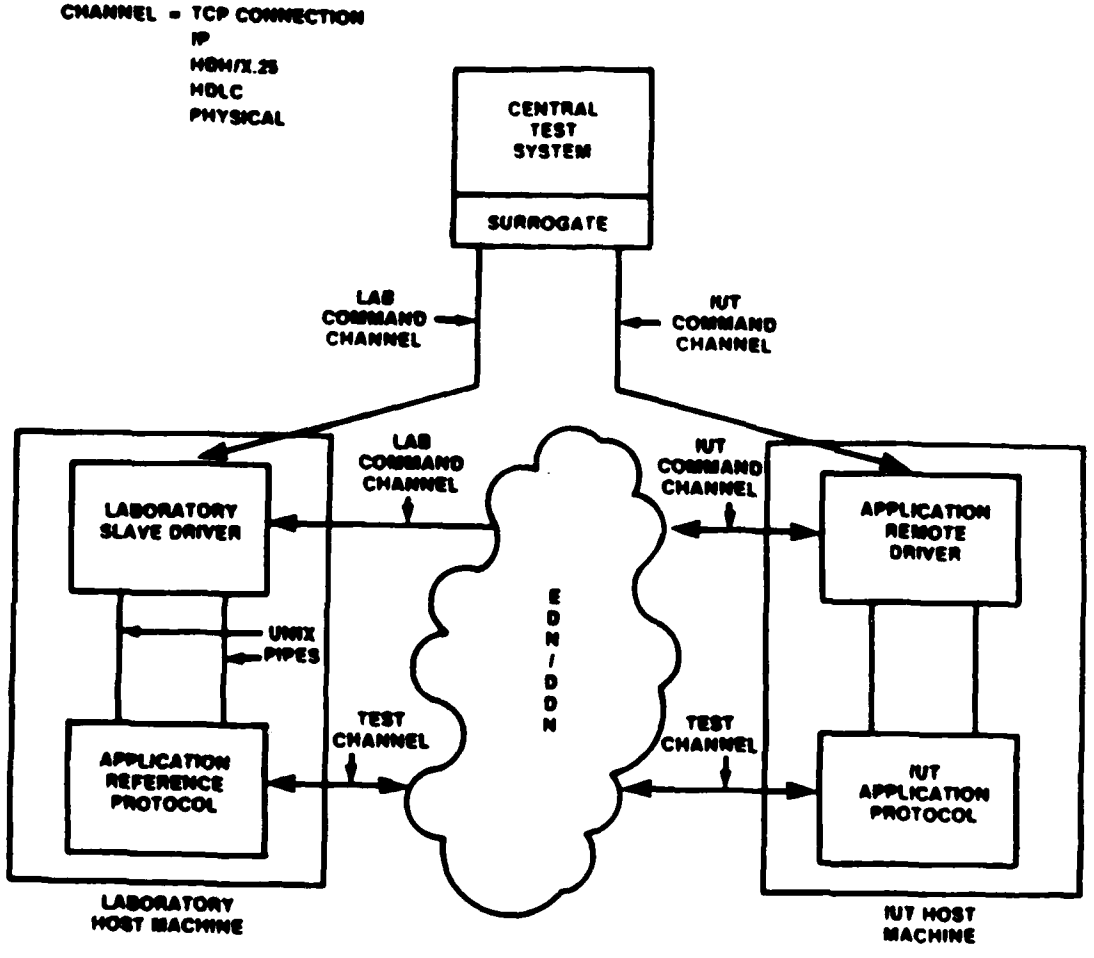

Figure 3-2. Flow of Commands Between the Four Types of Drivers

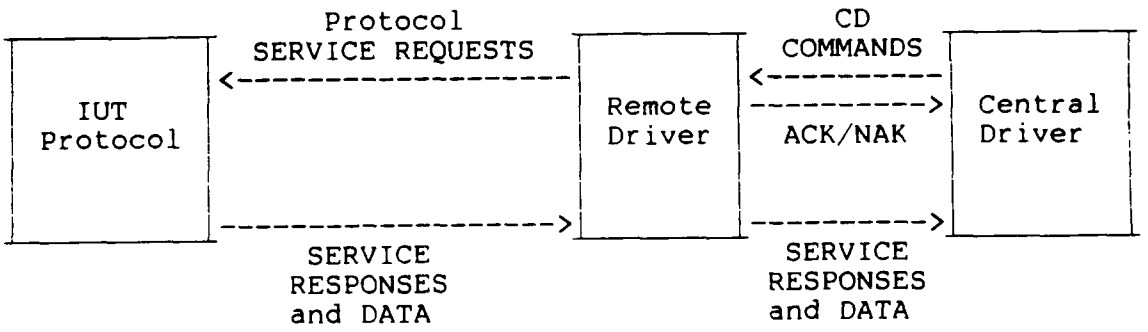

Figure 3-3. Remote Driver Functions

characters. The RD must also be able to send one of three packets: an ACK (Positive Acknowledgment), a NAK (Negative Acknowledgment), or a data packet containing either protocol responses or driver command results. The RD is required to acknowledge the receipt of every driver or protocol command. When the RD is able to read and interpret a packet from the CD, it responds by sending an ACK packet. If it is unable to read the packet, the RD responds by sending a NAK packet.

All codes are explained in detail in the following sections. Figure 3-5 (p. 3-7) shows the structure of the packets on the command channel; Figure C-2 in Appendix C defines a packet in C syntax. The packet and each of its fields are ordered in Internet Protocol (IP) byte order.

3.3.1 The Control Flag Field

The control flag, which is a single octet, indicates by bit position how the RD should interpret the rest of the data packet. The bit positions are in ascending order from right to left, as in a Digital Equipment Corporation (DEC) VAX. Figure 3-4 diagrams the bit-ordering scheme.

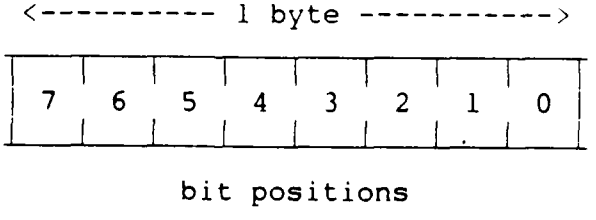

Figure 3-4. The Bit Order of a Byte

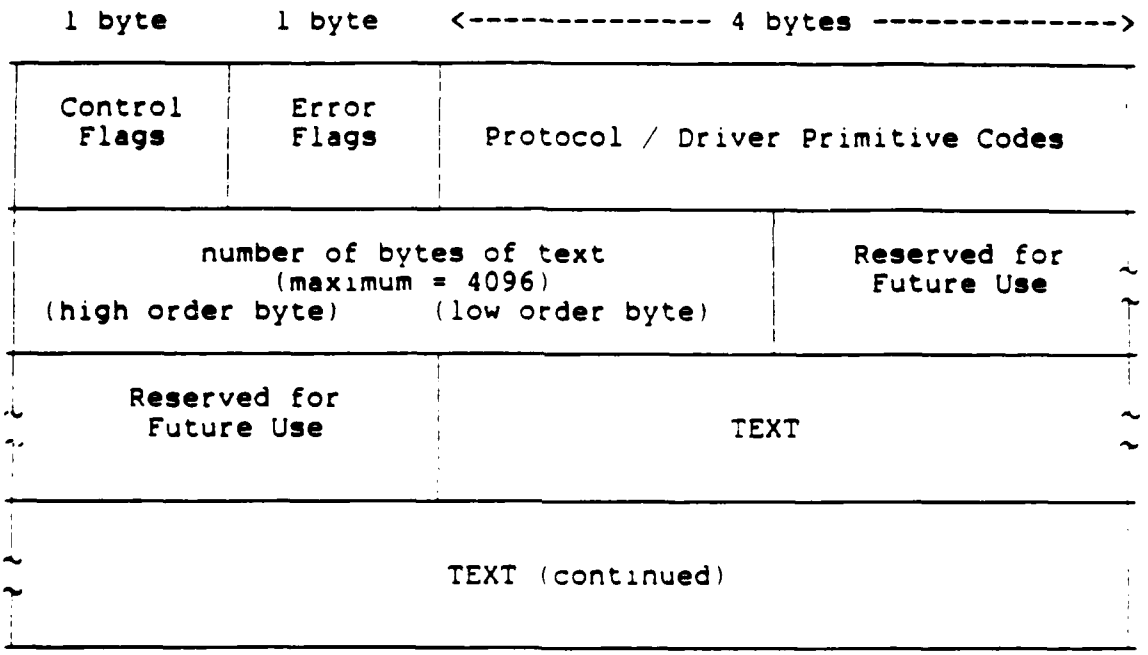

Figure 3-5. Structure of the Data Packet on the Command Channel

Table 3-2 summarizes the configuration of bit positions in the control flag. Only bit positions 0, 1, and 2 are used. The RD sets the bit in position 0 to one (1) to indicate that an ACK packet is being sent. If the RD sets the bit in position 0 to zero (0), the packet being transmitted is a NAK. Finally, the RD signals the transmission of a data packet by setting the bit in position 1 to one (1). Setting the data bit in a packet could signal either that the RD is sending a protocol response, or that it is sending the results of one of its own driver operations. The bit in position 2 of the first octet of each data packet sent by the CD determines whether the packet contains a protocol or a driver command.

Table 3-2. Bit Positions in the Control Flag

Bit Position	Remote-->Central	Central-->Remote
0	ACK = 1 NAK = 0	unused
1	data = 1	unused
2	unused	protocol command = 1 driver command = 0
3-7	unused	unused

3.3.2 The Error Flag Field

Error flags will explain the nature of an error occurring at the RD. Because error codes are not yet implemented, the protocol test system makes no attempt to interpret this field.

3.3.3 The Primitive Code Field

The primitive code field can be interpreted in one of two ways: either as a protocol primitive code or as a driver primitive code. A primitive in this sense means a command that describes some action within the protocol or the driver.

3.3.3.1 The Protocol Primitive Codes

If the control flag indicates a protocol command -- i.e., the bit in position 2 is set to one (1) -- then the RD must translate the integer in the code field to its corresponding protocol primitive according to the descriptions and the code numbers in Table 3-3. The format of the data sent by the RD to the protocol IUT depends on the IUT's Sender interface, but the RD must include this translation capability in its function. To pass qualification testing, an SMTP IUT must be able to perform all the primitive actions described in Table 3-3.

The RD determines whether arguments are associated with a primitive according to its protocol IUT. If arguments are required for a given primitive, they will be supplied in the data field of the data packet. If required arguments are not supplied, then an error condition occurs and the RD must NAK that packet.

The RD can determine whether the required arguments are present by reading the num_bytes field, which tells how many bytes of ASCII data are in the data field. If this field contains a zero, then the RD assumes no arguments have been supplied. However, if the num_bytes field contains any positive integer

from 0 to 4096, the RD must read the contents of the data field and interpret these contents as the primitive's arguments. An integer outside the range of 0 to 4096 means the packet is in error, and that a NAK packet should be returned.

In Table 3-3 the primitive commands and number codes are followed by one or more argument fields. Arguments are located in the data field of the command packet, beginning at location zero and separated by ASCII spaces. Three hyphens (---) in the table's argument column indicate that none exist. Arguments within square brackets are optional (e.g., [sender_name]).

Table 3-3. Protocol Primitive Commands, Number Codes, and Arguments, with Consequent SMTP IUT Actions (p. 1 of 3)

Primitive (**No. Code**)	Argument	SMTP IUT Action
helo **(1)**	[sender_name]	Same as MIL-STD-1781. Causes the Sender SMTP to send the HELO command over the command connection. The optional argument field is the Sender's identity.
mail **(2)**	[reverse_path]	Same as MIL-STD-1781. Causes the Sender SMTP to send the MAIL command over the command connection. The argument field identifies the Sender's reverse_path. Initiates a mail transaction in which the mail data are delivered to one or more mailboxes.
rcpt **(3)**	forward_path	Same as MIL-STD-1781. Causes the Sender SMTP to send the RCPT command over the command connection. The argument field identifies the forward_path to the destination mailbox.
data **(4)**	---	Same as MIL-STD-1781. Causes the Sender SMTP to send the DATA command over the command connection. Causes the Receiver SMTP to enter the state of accepting message data.
rset **(5)**	---	Same as MIL-STD-1781. The current mail transaction is aborted. Any stored sender, recipient, and mail data must be discarded; and all buffers and state tables are cleared.

(Table 3-3, p. 2 of 3)

Primitive (**No. Code**)	Argument	SMTP IUT Action
send **(6)**	[reverse_path]	Same as MIL-STD-1781. Causes the Sender SMTP to issue the SEND command over the command connection. The argument field identifies the Sender's reverse_path. Initiates a mail transaction in which the mail data are delivered to one or more terminals.
soml **(7)**	[reverse_path]	Same as MIL-STD-1781. Causes the Sender SMTP to issue the SOML command over the command connection. The argument field identifies the Sender's reverse_path. Initiates a mail transaction in which the mail data are delivered to one or more terminals or mailboxes.
saml **(8)**	[reverse_path]	Same as MIL-STD-1781. Causes the Sender SMTP to issue the SAML command over the command connection. The argument field identifies the Sender's reverse_path. Initiates a mail transaction in which the mail data are delivered to one or more terminals and mailboxes.
vrfy **(9)**	username	Same as MIL-STD-1781. Causes the Sender SMTP to send the VRFY command over the command connection. The Receiver confirms that the argument indicates a valid username. If valid, the complete name of the user (if known) and the fully specified mailbox are returned.

(Table 3-3, p. 3 of 3)

Primitive (**No. Code**)	Argument	SMTP IUT Action
expn **(10)**	mailing_list	Same as MIL-STD-1781. Causes the Sender SMTP to send the EXPN command over the command connection. The Receiver confirms that the argument indicates a valid mailing_list. If the mailing list is valid, the full membership of that list (if known) is returned in a multiline reply as user names and fully specified mailboxes.
help **(11)**	[cmd_string]	Causes the Sender SMTP to send the HELP command over the command connection. This command causes the Receiver to send helpful information regarding its implementation over the command connection to the Sender. The command may take an argument (e.g., any command name) and return more specific information as a response.
noop **(12)**	---	Same as MIL-STD-1781. Causes the Sender SMTP to send the NOOP command over the command connection. This command does not affect any parameters or previously entered commands. It specifies no action, but the Receiver must send an OK reply.
quit **(13)**	---	Same as MIL-STD-1781. Causes the Sender SMTP to send the QUIT command over the command connection. The Receiver must send an OK reply, then close the transmission channel.
turn **(14)**	---	Same as MIL-STD-1781. Causes the Sender SMTP to send the TURN command over the command connection. This command causes the Sender and Receiver SMTP to switch roles if the Receiver agrees.

3.3.3.2 The Driver Primitive Codes

If the control flag indicates a driver command (the bit in position 2 is set to zero), the RD must translate the integer contained in the code field to its corresponding driver primitive (Table 3-4). The RD then performs the appropriate action, as specified in the following sections.

In Table 3-4, the primitive commands and number codes are followed by one or more argument fields. Arguments are located in the data field of the command packet, beginning at location zero and separated by ASCII spaces. Three hyphens (---) in the table's argument column indicate that none exist. Arguments inside square brackets ([ ]) are optional. If arguments are within braces and separated by vertical bars, then one of the arguments is mandatory (e.g., { forward_path | mailing_list }).

3.3.3.2.1 The DIE Driver Primitive

If the RD receives a driver primitive code of **31**, indicating the DIE command, after ACKing the driver primitive packet, the RD must close the test connection to the CD, close all connections to the IUT, and halt the RD process. This command does not send any data across the test connection after the initial ACK packet.

3.3.3.2.2 The CLOSE Driver Primitive

If the RD receives a driver primitive code of **32**, indicating the CLOSE command, after ACKing the driver primitive packet, the RD must close the test connection to the CD, close all connections to the IUT (RD-to-IUT connections are implementation dependent),

and wait for a new connection attempt from the CD. The RD process does not die as in the DIE command; it waits for a new connection. This command does not send any data across the test connection after the initial ACK packet.

Table 3-4. The Driver Primitive Commands, Number Codes, and Arguments, with Consequent Remote Driver Actions (p. 1 of 2)

Driver Primitive (**No. Code**)	Argument	Remote Driver Action
die **(31)**	---	Causes the RD to close the test connection to the CD, close all connections to the IUT, and die. This command does not send any data across the command connection. (Section 3.3.3.2.1 gives a detailed description.)
close **(32)**	---	Causes the RD to close the test connection to the CD, close all connections to the IUT, and wait for a new connection to the CD. This command does not send any data across the command connection. (Section 3.3.3.2.2 gives a detailed description.)
spool **(41)**	username	Causes the RD to send all the received mail messages for the specified user over the command channel to the CD. If the RD is unable to spool the mailbox, it sends the message "!!ERROR!!" to the CD. The end of the mail message is indicated by the string "END_OF_SPOOL". (Section 3.3.3.2.3 gives details.)
delete **(42)**	username	Causes the RD to delete all the received mail messages for the specified user from the user's mailbox. If the RD is unable to delete the mailbox, it sends the message "!!ERROR!!" to the CD. (Section 3.3.3.2.4 gives details.)

(Table 3-4, p.2 of 2)

Driver Primitive (**No. Code**)	Argument	Remote Driver Action
queue **(43)**	delivery username { forward_path | mailing_list } [message]	Causes the RD to send message from specified username to forward_path; or list of forward paths represented by mailing_list. The delivery parameter indicates one of the SMTP methods of mail delivery (MAIL, SEND, SAML, or SOML). If the message field is not present, the contents of the specified user's mailbox are sent to forward_path. If the RD is unable to queue the message or the mailbox, it replies "!!ERROR!!" to the CD. The end of the mail message is indicated by the string "END_OF_SPOOL". (See Section 3.3.3.2.5 for a detailed description.)

3.3.3.2.3 The SPOOL Driver Primitive

If the RD receives a driver primitive code of **41**, indicating the SPOOL command, after ACKing the driver primitive packet, the RD must send all the received mail messages for the specified user over the test connection to the CD. When the RD has finished sending all the mail data (in data packets), it generates a string "END_OF_SPOOL" to indicate end of data. The string is sent as 7-bit ASCII characters. The underbar character ("_") is octal ASCII character 137. If the RD cannot send the mail data for any reason, after ACKing the command, it must send the string "!!ERROR!!" in a data packet to the CD. The CD performs a string search on the data it receives and determines the success or failure of a test according to the results of the search, so care should be taken to return the data accurately.

Example:

```
spool "test1"
```

3.3.3.2.4 The DELETE Driver Primitive

If the RD receives a driver primitive code of **42**, indicating the DELETE command, after ACKing the driver primitive packet, the RD must delete all the received mail messages for the specified user from the user's mailbox.

Example:

```
delete "test1"
```

3.3.3.2.5 The QUEUE Driver Primitive

If the RD receives a driver primitive code of **43**, indicating the QUEUE command, after ACKing the driver primitive packet, the RD must cause the IUT to deliver the message from the user specified by the username parameter to the forward_path or to the list of forward_paths represented by mailing_list. The method of delivery is specified by the delivery parameter. If the message field is not present, then the contents of the user's mailbox is sent to the forward_path. If the RD is unable to delete the mail data for any reason, after ACKing the command, it must send the string "!!ERROR!!" in a data packet to the CD. The CD performs a string search on the data it receives and determines the success or failure of a test according to the results of the search, so care should be taken to send back the data accurately. The fields are explained in detail in the following paragraphs.

The delivery field indicates which SMTP method of delivery will be used for message transmission. There are only four possibilities: MAIL, SEND, SAML, or SOML.

The second field is the username field, which is a character string containing a valid source mailbox.

The third field contains either a forward_path or a mailing_list. A forward_path describes a destination mailbox and how to get there. The syntax of the forward_path must follow MIL-STD-1781.

A mailing list is a list of forward paths. It is assumed that "mailing_list" is a local list at the IUT-host and that the delivery of the mail messages will be performed by multiple RCPT commands.

The fourth field is the optional message field. If no message is specified, the contents of the source mailbox specified in the username field will be used as the mail message. If the message is specified, the contents of the string -- contained within double quotes -- will be used as the contents of the mail message.

The following examples of some possible uses of the QUEUE driver primitive may help explain its function. These examples are excerpts taken from a typical test script. (Test scripts define different scenarios.) The RD would receive the number code for the QUEUE driver primitive (**43**), and translate the number into the appropriate QUEUE command function using the corresponding parameters supplied with the driver primitive.

Examples:

```
queue "MAIL test1 <test2@protolabb> Test_message_goes_here"
```

This command transmits a mail message from <test1@IUT-host> to <test2@protolabb>; "IUT-host" is the hostname where the IUT resides. The message is the string "Test_message_goes_here"; and delivery of the message is via the SMTP MAIL command.

queue "SAML testl <test3@protolabb> testing_123"

This command transmits a message from <testl@IUT-host> to <test3@protolabb>. The mail message data consist of the string "testing_123"; delivery is via the SMTP SAML command.

queue "SEND testl <test2@protolabb>"

This command transmits a message from <test2@protolabb> to the terminal at forward_path <test2@protolabb>, if the user "test2" is logged on at the time. The SMTP SEND command is specified as the method of delivery, so the message must be delivered to the destination terminal. Since no message parameter is specified, the message consists of the contents of the mailbox of the user "testl".

queue "SAML testl testlist testing_123"

This command transmits a mail message from <testl@IUT-host> to the members of the mailing list "testlist". It is assumed that "testlist" is a local list at the IUT-host and that the delivery of the mail messages, via the SAML command, will be performed by multiple RCPT commands. The mail message data consist of the string "testing_123".

3.4 ACK/NAK PACKETS

The RD is required to acknowledge the receipt of every driver or protocol command (ACK = positive acknowledgment; NAK = negative acknowledgment). When it is able to read and interpret a packet from the TCP connection, the RD sends an ACK packet. If it detects an error, however, the RD responds with a NAK packet. The RD also sends a NAK packet if a read is successful but a code is unknown; or if some other field is in error.

The following values indicate ACK or NAK packets:

ACK

code : **0**

cntl_flag : bit position 0 set to one (**1**)
bit position 1 is unused
bit position 2 set to one (**1**) if protocol command, to zero (**0**) if driver command

num_bytes : **0**

data : **empty**

NAK

code : **0**

cntl_flag : bit position 0 set to zero (**0**)
bit position 1 is unused
bit position 2 set to one (**1**) if protocol command, to zero (**0**) if driver command

num_bytes : **0**

data : **empty**

3.5 TIMING

An RD has no intrinsic timing constraints, but it should not add considerably to a protocol IUT's response time. For example, if the CD does not receive a data packet within the time specified in a script, then a timeout condition will occur. Such a condition could cause an IUT to fail the test.

3.6 FLEXIBILITY

Although RDs have no special flexibility requirements, adaptable hardware and software enhance their operation and expandability.

APPENDIX A - References

"Military Standard Transmission Control Protocol (MIL-STD-1778); August 1983; Department of Defense.

"Military Standard Internet Protocol" (MIL-STD-1777); August 1983; Department of Defense.

System Development Corporation, "Laboratory Implementation Plan, "TM-WD-857/000/02, January 1985.

System Development Corporation, "Laboratory Specification," TM-WD-7172/520/00, August 1984.

System Development Corporation, "Higher Level Capability Plan," TM-WD-857/000/00, April 1984.

Kernighan, B. W., and Ritchie, D. M.; The C Programming Language; Prentice-Hall, Inc.; Englewood Cliffs, NJ; 1978.

Kernighan, B. W., and Pike, R.; The UNIX Programming Environment; Prentice-Hall, Inc.; Englewood Cliffs, NJ; 1984.

APPENDIX B - Glossary

ACK (Acknowledgment)

In data transfer between devices, data are blocked into units of a size given in each block's header. If the received data are found to be without errors, the receiving device sends an ACK block back to the transmitting unit to acknowledge receipt. The transmitting device then sends the next block. If the receiving unit detects errors it sends a NAK block to indicate the received data contained errors.

ASCII (American Standard Code for Information Interchange)

A standard code for the representation of alphanumeric information. ASCII is an 8-bit code in which 7 bits indicate the character represented and the 8th, high-order bit is used for parity.

CD (Central Driver)

DDN (Defense Data Network)

A Department of Defense packet-switched network.

DEC (Digital Equipment Corporation)

DoD (Department of Defense)

EDN (Exploratory Data Network)

IUT (Implementation Under Test)

A given vendor's protocol implementation and the subject of the immediate test.

MIL-STD (Military Standard)

Specification published by the DoD.

NAK (Negative Acknowledgment)

In data transfer between devices, a NAK block is returned by the receiving device to the sending device to indicate the preceding data block contained errors. See also ACK.

packet switching

A method of transmitting messages through a network in which long messages are subdivided into short packets. The packets are then transmitted as in message switching.

PAD (Packet Assembler/Disassembler)

In this document, PAD refers to a module of a structured program responsible for reading and writing data packets. Not to be confused with the other known usage, which describes a device to provide service to asynchronous terminals within an X.25 network.

PAR (Positive Acknowledgment and Response)

A simple communication protocol stating that every packet received must be either ACKed or NAKed.

protocol

A set of rules governing the operation of functional units to achieve communication.

TCP (Transmission Control Protocol)

The DoD standard connection-oriented transport protocol used to provide reliable, sequenced, end-to-end service.

ULP (Upper Level Protocol)

Any protocol above TCP in the layered protocol hierarchy that uses TCP. This term includes presentation layer protocols, session layer protocols, and user applications such as the protocol test system's drivers.

APPENDIX C - Examples of Remote Driver Implementation in UNIX/C

C.1 CONNECTION ESTABLISHMENT

The protocol laboratory runs in a UNIX 4.2 BSD environment. The lab's example RD is implemented in the C language, which provides access to several interprocess communication system calls (e.g., the fork, socket, bind, listen, and accept system calls).

The design of the RD uses the Internet Server model. The first step taken by the RD is to disassociate itself from the controlling terminal of the invoker. It then does a passive open and listens at the published well-known port for a connection request from the Surrogate Driver. Using the fork system call, the RD creates a copy of itself (a child process), so that it can continue to listen for connection requests following a successful connection. It is this child process that carries out the functions of the RD, communicating over the established command channel. The original, or parent, process remains in execution and listens to the well-known port. Figure C-1 outlines the establishment of the command channel in 4.2 BSD UNIX/C. In addition to the UNIX system calls, the example uses 4.2 BSD library calls to eliminate direct handling of Internet protocol numbers and addressing.

C.2 A PAD FUNCTION

A Packet Assembler/Disassembler (PAD) function is useful for implementing an RD. Because the two basic functions of receiving and transmitting packets are performed repeatedly,

```
/*                  COMMAND CHANNEL ESTABLISHMENT
 */
struct protoent *pp;

struct servent *sp;

int socket_number, socket_number2;

pp = getprotobyname("tcp")          /* 4.2 BSD library call */

sp = getservbyname("remote_driver", "tcp");*/ 4.2 BSD
    library call
    */
#ifndef DEBUG
    /*

<< disassociate RD from controlling terminal here>>

    */
#endif
    :
    :
sin.sin_port = sp->s_port; /* set port number */
    :
    :
socket_number = socket(AF_INET, SOCK_STREAM, pp->p_proto);

bind(socket_number, (caddr_t)&sin, sizeof(sin), 0);

listen(socket_number, 5);/* passive open to listen at
                         well-known port
                            */
for(;;) {                   /* do forever */

    socket_number2 = accept(socket_number, 0, 0);

    if (fork() == 0) {   /* this is the child process if
        true */ close(socket number);
                         /* socket_number2 is the local
                            connection
                          * name of the command channel
                          */
        do_remote_driver_functions(socket_number2);
    }
    close(socket_number2);/* the parent goes back to
        listening */
                         /* on socket_number            */
}
```

Figure C-1. Outline of Connection Establishment in 4.2 BSD UNIX/C

it may be wise to modularize them. When the RD reads data from the command channel, it must be able to interpret the bytes correctly. In a UNIX/C implementation, the method used to achieve this result is to load the data into a data structure where distinct fields can be declared. In C, this is the struct declaration. Each collection of fields can be declared and referenced as a whole. The term packet has been used in this document as the name of this data structure reference. The struct declaration is shown in Figure C-2.

```
#define  MAX_TEXT_LEN       4096
struct remote_pack  {
                char cntl_flag;
                char err_flag;
                int  code;
                int  num_bytes;
                int  reserved;
                char text[MAX_TEXT_LEN];
              };
```

Figure C-2. The C Syntax Format of the Data Packet

The reception mode of the PAD function reads and packetizes the data. The PAD accomplishes this task by reading the first 14 bytes of data from the input stream. These first 14 bytes effectively constitute the header of the data packet. The header contains all the information needed to process the packet, including the integer in the num_bytes field that indicates the number of bytes of character text, if any, to follow. If the input data stream is correctly packetized into a data structure, then interpretation of the packet's fields should become clearer and less susceptible to error.

The transmission mode of the PAD function depacketizes the data and sends it over the communication channel. The PAD accomplishes this task by writing the packet header and then the text into a memory space allocated for a large character string. The size is equal to 4096 bytes -- the maximum text size -- plus 14 bytes for the header information. If the text size is less than the maximum, then only that much need be written. The RD then transmits the data as a normal byte stream.

An example of a PAD implemented in C is given in Figure C-3 (p. C-5). The example is specific to UNIX/C on a VAX with 32-bit integers, where the routines for converting between host (VAX) word order and network word order are mandatory.

```
#define HEADER_SIZE   14 /* WARNING: VAX word-ordering */
#define MAX_TEXT_LEN  4096
#define MAX_PACK_LEN HEADER_SIZE + MAX_TEXT_LEN

send_packet(packet,sock)       /* packet disassembler      */
struct remote_pack *packet;
int sock;
{   register int i;
    char send_buffer[MAX_PACK_LEN];

    send_buffer[0] = packet->cntl_flag;
    send_buffer[1] = packet->err_flag;    /* host-network
        conversion */
    *((int *)(send_buffer+2))    = htonl(packet->code);
    *((int *)(send_buffer+6))    = htonl(packet->num_bytes);
    *((int *)(send_buffer+10))   = htonl(packet->reserved);
    for(i = 0; i < packet->num_bytes; i++)
         send_buffer[i+14] = packet->text[1];
                              /* send the packet          */
    return(write(sock, send_buffer,
                 packet->num_bytes+HEADER_SIZE));
}

recv_packet(packet,sock)          /* packet assembler      */
struct remote_pack *packet;
int sock;
{
    char recv_buffer[HEADER_SIZE];
    int result;
        /* read the header */
    if ((result = read(sock, recv_buffer, HEADER_SIZE)) !=
                                              HEADER_SIZE)
    {
        return (result);
    }
    packet->cntl_flag = recv_buffer[0];
    packet->err_flag  = recv_buffer[1];/* net-host
        conversion */
    packet->code = ntohl(*((int *) (recv_buffer+2)));
    packet->num_bytes = ntohl(*((int *) (recv_buffer+6)));
    packet->reserved = ntohl(*((int *) (recv_buffer+10)));
        /* read the text*/
    if (packet->num_bytes)
    {
        if(read(sock, packet->text, packet->num_bytes)
                                    =packet->num_bytes
        return(-1);
    }
    return(0);
}
```

Figure C-3. A Packet Assembler/Disassembler in C